

**CMM**

Center for  
Mathematical  
Modeling

# Redes neuronales convolucionales

Y problemas al entrenar redes neuronales

**Diego Olguín (basado en un trabajo conjunto con Javier Maass)**

Centro de Modelamiento Matemático  
Universidad de Chile

9 de septiembre

# Contenidos

**1. Entendiendo Problemas de las NN**

**2. Regularización en NNs**

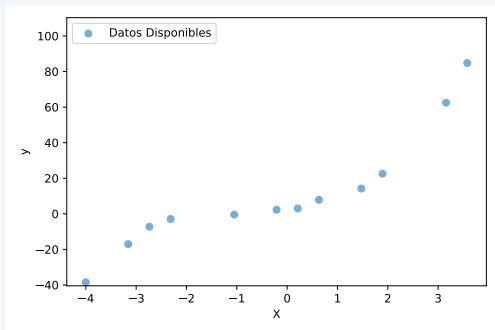
**3. Clasificación y Redes convolucionales**

# Entendiendo Problemas de las NN

---

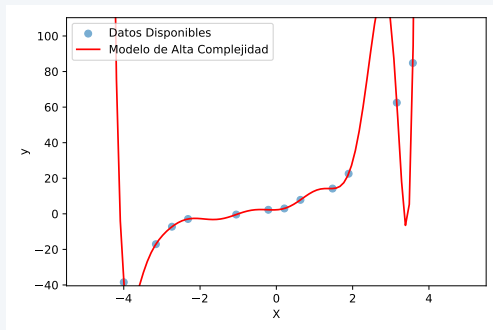
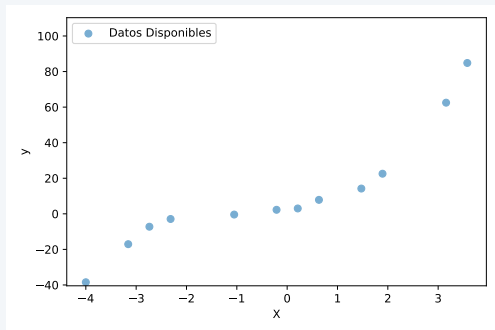
# Problemas en el Entrenamiento

Si usamos todos nuestros datos para entrenar un modelo, nos puede ir muy bien!



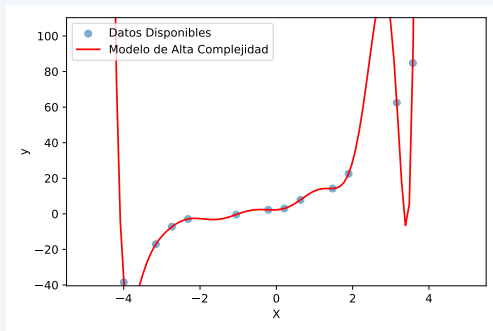
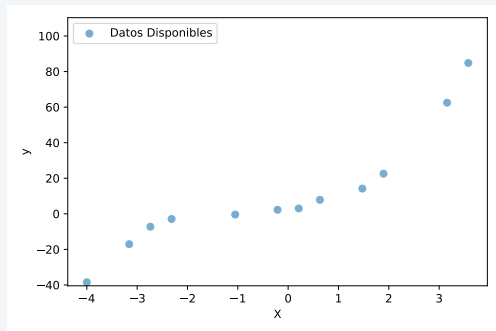
# Problemas en el Entrenamiento

Si usamos todos nuestros datos para entrenar un modelo, nos puede ir muy bien!



# Problemas en el Entrenamiento

Si usamos todos nuestros datos para entrenar un modelo, nos puede ir muy bien!



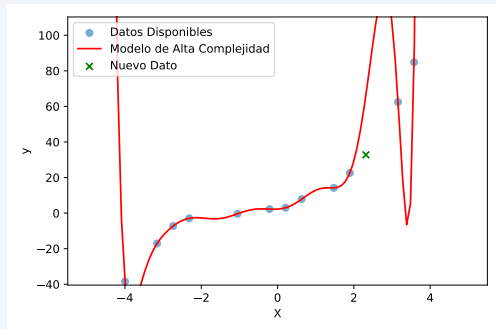
¿Hay algún problema con este modelo? ¿Cómo le irá si nos llegan datos nuevos?

# Problemas en el Entrenamiento

Si trabajamos de forma naïve, ¡muy mal!

# Problemas en el Entrenamiento

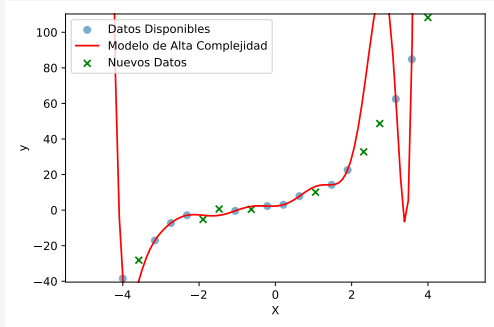
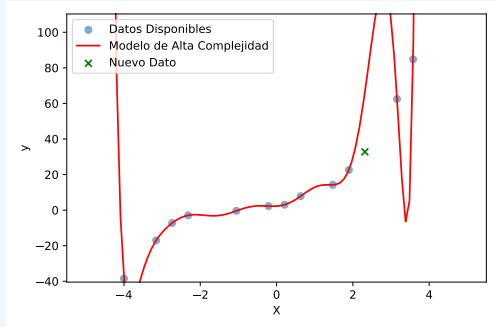
Si trabajamos de forma naïve, ¡muy mal!





# Problemas en el Entrenamiento

Si trabajamos de forma naïve, ¡muy mal!

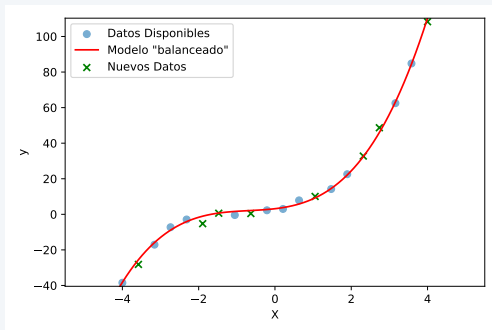


# Problemas en el Entrenamiento

A veces es mejor un modelo más simple para lograr *generalizar mejor*

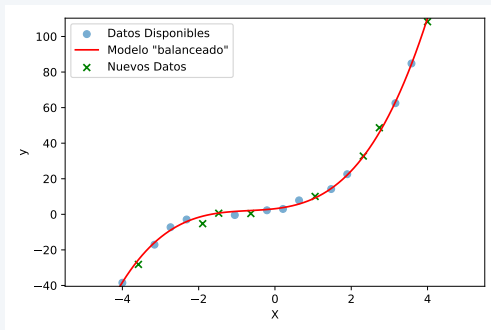
# Problemas en el Entrenamiento

A veces es mejor un modelo más simple para lograr *generalizar mejor*



# Problemas en el Entrenamiento

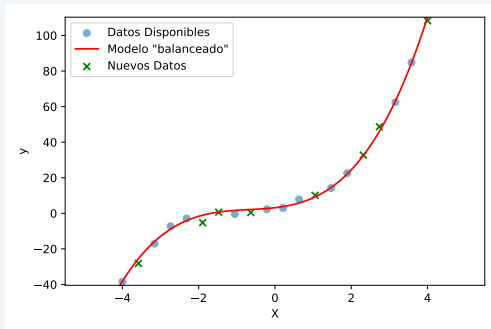
A veces es mejor un modelo más simple para lograr *generalizar mejor*



En general, queremos modelos que minimicen  $R(\theta) := \mathbb{E} [\mathcal{L}(\Phi_{\theta}(X), Y)]$

# Problemas en el Entrenamiento

A veces es mejor un modelo más simple para lograr *generalizar mejor*



En general, queremos modelos que minimicen  $R(\theta) := \mathbb{E} [\mathcal{L}(\Phi_{\theta}(X), Y)]$   
¿Cómo podemos estimar qué tan ‘buenos’ son nuestros modelos?

# Train/Test Split

Lo estándar es **separar los datos en conjuntos de Train, Test y (a veces) Valid**. Sea  $Z$  la ley conjunta de  $(X, Y)$  (*inputs* y *target*), con una ley de probabilidad  $\mathbb{P}_Z$ .

# Train/Test Split

Lo estándar es **separar los datos en conjuntos de Train, Test y (a veces) Valid**. Sea  $Z$  la ley conjunta de  $(X, Y)$  (*inputs* y *target*), con una ley de probabilidad  $\mathbb{P}_Z$ . Si  $s = \{(x_i, y_i)\}_{i=1}^m$  y  $\tilde{s} = \{(\tilde{x}_i, \tilde{y}_i)\}_{i=1}^{\tilde{m}}$  son train y test resp. (i.i.d.  $\sim \mathbb{P}_Z$ ):

# Train/Test Split

Lo estándar es **separar los datos en conjuntos de Train, Test y (a veces) Valid**. Sea  $Z$  la ley conjunta de  $(X, Y)$  (*inputs* y *target*), con una ley de probabilidad  $\mathbb{P}_Z$ . Si  $s = \{(x_i, y_i)\}_{i=1}^m$  y  $\tilde{s} = \{(\tilde{x}_i, \tilde{y}_i)\}_{i=1}^{\tilde{m}}$  son train y test resp. (i.i.d.  $\sim \mathbb{P}_Z$ ):

- Al entrenar, minimizamos:  $\hat{R}^{train} = \hat{R}_s(\theta) := \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\Phi_\theta(x_i), y_i)$  para obtener  $\theta_s^*$ .



# Train/Test Split

Lo estándar es **separar los datos en conjuntos de Train, Test y (a veces) Valid**. Sea  $Z$  la ley conjunta de  $(X, Y)$  (*inputs* y *target*), con una ley de probabilidad  $\mathbb{P}_Z$ . Si  $s = \{(x_i, y_i)\}_{i=1}^m$  y  $\tilde{s} = \{(\tilde{x}_i, \tilde{y}_i)\}_{i=1}^{\tilde{m}}$  son train y test resp. (i.i.d.  $\sim \mathbb{P}_Z$ ):

- Al entrenar, minimizamos:  $\hat{R}^{train} = \hat{R}_s(\theta) := \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\Phi_\theta(x_i), y_i)$  para obtener  $\theta_s^*$ .
- Para aproximar el verdadero riesgo de población, utilizamos los **datos de test**, con esto vemos el desempeño del modelo en puntos no vistos:

$$\hat{R}^{test} := \hat{R}_{\tilde{s}}(\theta_s^*) := \frac{1}{\tilde{m}} \sum_{i=1}^{\tilde{m}} L(\Phi_{\theta_s^*}(\tilde{x}_i), \tilde{y}_i) \approx R(\theta_s^*)$$

# Train/Test Split

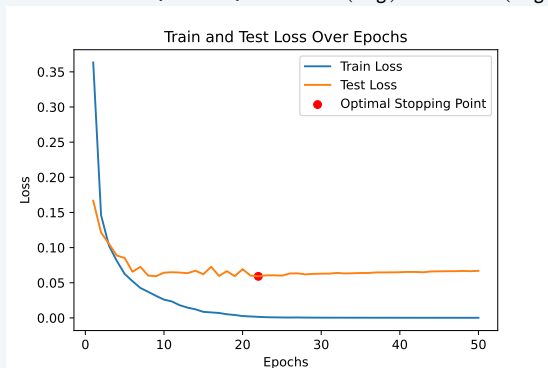
Lo estándar es **separar los datos en conjuntos de Train, Test y (a veces) Valid**. Sea  $Z$  la ley conjunta de  $(X, Y)$  (*inputs* y *target*), con una ley de probabilidad  $\mathbb{P}_Z$ . Si  $s = \{(x_i, y_i)\}_{i=1}^m$  y  $\tilde{s} = \{(\tilde{x}_i, \tilde{y}_i)\}_{i=1}^{\tilde{m}}$  son train y test resp. (i.i.d.  $\sim \mathbb{P}_Z$ ):

- Al entrenar, minimizamos:  $\hat{R}^{train} = \hat{R}_s(\theta) := \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\Phi_\theta(x_i), y_i)$  para obtener  $\theta_s^*$ .
- Para aproximar el verdadero riesgo de población, utilizamos los **datos de test**, con esto vemos el desempeño del modelo en puntos no vistos:

$$\hat{R}^{test} := \hat{R}_{\tilde{s}}(\theta_s^*) := \frac{1}{\tilde{m}} \sum_{i=1}^{\tilde{m}} L(\Phi_{\theta_s^*}(\tilde{x}_i), \tilde{y}_i) \approx R(\theta_s^*)$$

# Train/test split

Al entrenar una NN, en general, evaluamos cómo evolucionó la loss durante el entrenamiento. En general, se espera que:  $\hat{R}^{\text{test}}(\Theta_s^*) \geq \hat{R}^{\text{train}}(\Theta_s^*)$ .



# Underfitting y Overfitting

- **Underfitting:**  $\uparrow \hat{R}^{train}$  y  $\uparrow \hat{R}^{test}$ . i.e. el **modelo no tiene capacidad suficiente**.

# Underfitting y Overfitting

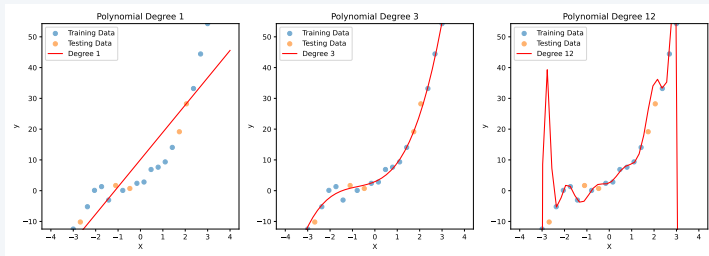
- **Underfitting:**  $\uparrow \hat{R}^{train}$  y  $\uparrow \hat{R}^{test}$ . i.e. el **modelo no tiene capacidad suficiente**.
- **Overfitting:**  $\downarrow \hat{R}^{train}$ , pero  $\uparrow \hat{R}^{test}$ . i.e. el modelo *memoriza* el conjunto de train y **no logra generalizar bien** en datos nuevos.

# Underfitting y Overfitting

- **Underfitting:**  $\uparrow \hat{R}^{train}$  y  $\uparrow \hat{R}^{test}$ . i.e. el **modelo no tiene capacidad suficiente**.
- **Overfitting:**  $\downarrow \hat{R}^{train}$ , pero  $\uparrow \hat{R}^{test}$ . i.e. el modelo *memoriza* el conjunto de train y **no logra generalizar bien** en datos nuevos.
- **Modelo bien ajustado:**  $\downarrow \hat{R}^{train}$  y  $\downarrow \hat{R}^{test}$ . i.e. **modelo aprende y generaliza**.

# Underfitting y Overfitting

- **Underfitting:**  $\uparrow \hat{R}^{train}$  y  $\uparrow \hat{R}^{test}$ . i.e. el **modelo no tiene capacidad suficiente**.
- **Overfitting:**  $\downarrow \hat{R}^{train}$ , pero  $\uparrow \hat{R}^{test}$ . i.e. el modelo *memoriza* el conjunto de train y **no logra generalizar bien** en datos nuevos.
- **Modelo bien ajustado:**  $\downarrow \hat{R}^{train}$  y  $\downarrow \hat{R}^{test}$ . i.e. **modelo aprende y generaliza**.



# Regularización en NNs

---



# Regularización

Nos interesa aplicar técnicas que nos permitan asegurar que las NN sólo aprenderán **características relevantes** y no se *sobreajustarán* al problema.

# Regularización

Nos interesa aplicar técnicas que nos permitan asegurar que las NN sólo aprenderán **características relevantes** y no se *sobreajustarán* al problema.

En la práctica se usan varias. Aquí veremos en detalle:

- Regularización Clásica: L1 y L2.
- *Dropout*.
- Otras (*Batch Normalization, Data Augmentation, Sparse Networks*, etc.).

# Regularización Clásica

El *overfitting* de una NN podría deberse a la *explosión* de sus parámetros.

# Regularización Clásica

El *overfitting* de una NN podría deberse a la *explosión* de sus parámetros. Podemos intentar mitigar esto introduciendo una *penalización* en el problema:

# Regularización Clásica

El *overfitting* de una NN podría deberse a la *explosión* de sus parámetros. Podemos intentar mitigar esto introduciendo una *penalización* en el problema:

## Definición (Regularización Clásica)

Sea  $r : \Theta \rightarrow \mathbb{R}$  una función de *penalización* y  $\lambda > 0$ ; intentaremos resolver:

$$\min_{\theta \in \Theta} \hat{R}_s(\theta) + \lambda \cdot r(\theta)$$

Cuando  $r = \|\cdot\|_2$  ( $\|\cdot\|_1$  resp.) lo llamamos regularización L2 (L1 resp.).

# Regularización Clásica

El *overfitting* de una NN podría deberse a la *explosión* de sus parámetros. Podemos intentar mitigar esto introduciendo una *penalización* en el problema:

## Definición (Regularización Clásica)

Sea  $r : \Theta \rightarrow \mathbb{R}$  una función de *penalización* y  $\lambda > 0$ ; intentaremos resolver:

$$\min_{\theta \in \Theta} \hat{R}_s(\theta) + \lambda \cdot r(\theta)$$

Cuando  $r = \|\cdot\|_2$  ( $\|\cdot\|_1$  resp.) lo llamamos regularización L2 (L1 resp.).

La regularización L1 tiende a lograr que los parámetros se hagan 0, logrando con ello una mayor *sparsity* en la red.

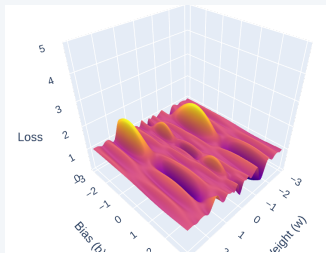
# Regularización Clásica

e.g. Con **regularización L2**:

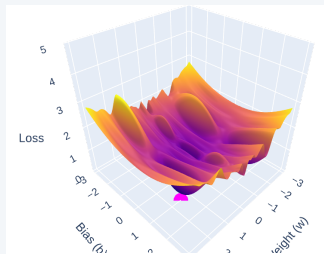
# Regularización Clásica

e.g. Con **regularización L2**:

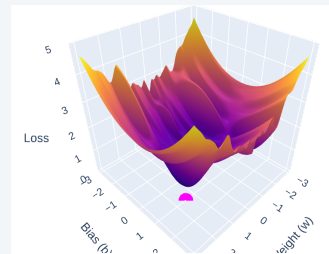
$$\lambda = 0$$



$$\lambda = 0,1$$



$$\lambda = 0,2$$





# Dropout

Dropout previene el overfitting en NNs evitando *co-dependencias* entre neuronas.

# Dropout

Dropout previene el overfitting en NNs evitando *co-dependencias* entre neuronas.

**Idea:** Al entrenar, se eliminan aleatoriamente ciertas neuronas ocultas de la red.

# Dropout

Dropout previene el overfitting en NNs evitando *co-dependencias* entre neuronas.

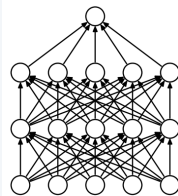
**Idea:** Al entrenar, se eliminan aleatoriamente ciertas neuronas ocultas de la red.

## Definición (Dropout)

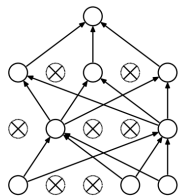
Sea  $\Phi_{\Theta}$  una NN de  $L$  capas, con  $h_{(l)} \in \mathbb{R}^{n_l}$  la salida de la capa  $l \in [L]$ . Definimos, para  $(p_l)_{l=1}^L \subseteq (0, 1]$ :

$$\text{Dropout}(h_{(l)}) = h^{(l)} \odot m_{(l)}$$

con  $\odot$  multiplicación *pointwise*, y  $m_{(l)} \in \{0, 1\}^{n_l}$  es una máscara aleatoria:  $(m_{(l)})_i \sim \text{Bernoulli}(p_l)$ .



(a) Standard Neural Net



(b) After applying dropout.

# Dropout

Dropout previene el overfitting en NNs evitando *co-dependencias* entre neuronas.

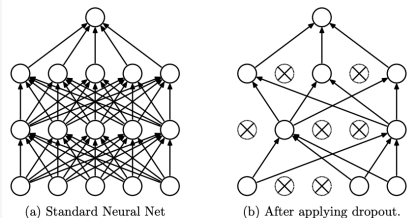
**Idea:** Al entrenar, se eliminan aleatoriamente ciertas neuronas ocultas de la red.

## Definición (Dropout)

Sea  $\Phi_{\Theta}$  una NN de  $L$  capas, con  $h_{(l)} \in \mathbb{R}^{n_l}$  la salida de la capa  $l \in [L]$ . Definimos, para  $(p_l)_{l=1}^L \subseteq (0, 1]$ :

$$\text{Dropout}(h_{(l)}) = h^{(l)} \odot m_{(l)}$$

con  $\odot$  multiplicación *pointwise*, y  $m_{(l)} \in \{0, 1\}^{n_l}$  es una máscara aleatoria:  $(m_{(l)})_i \sim \text{Bernoulli}(p_l)$ .



Al entrenar, esto se aplica a la salida de cada capa; en inferencia, se usa la red completa, pero escalando las salidas por  $p$ .

## Dropout y regularización L2

Veamos el caso de la regresión lineal: sea  $X \in \mathbb{R}^{N \times D}$  una matriz de datos,  $y \in \mathbb{R}^N$  un vector de objetivos. Recordar que la regresión lineal busca un  $w \in \mathbb{R}^D$  que minimice

$$\|y - Xw\|^2.$$

Cuando la entrada  $X$  se somete a *dropout*, de manera que cada dimensión de la entrada se retiene con probabilidad  $p$ , la entrada puede expresarse como  $R * X$ , donde  $R \in \{0, 1\}^{N \times D}$  es una matriz aleatoria con  $R_{ij} \sim \text{Bernoulli}(p)$  y  $*$  denota el producto elemento a elemento. Al marginalizar el ruido, la función objetivo se convierte en

$$\min_w \mathbb{E}_{R \sim \text{Bernoulli}(p)} [\|y - (R * X)w\|^2].$$

# Dropout y regularización L2

## Teorema

*La regresión lineal con dropout se reduce a*

$$\min_w \|y - pXw\|^2 + p(1 - p) \|\Gamma w\|^2,$$

*es decir, es equivalente a una regularización L2.*

# Clasificación y Redes convolucionales

---

# Problemas de Clasificación

**Recuerdo:** Buscamos predecir *etiquetas*  $y \in \mathcal{C}$ , con  $\mathcal{C}$  finito (digamos,  $\mathcal{C} = [K]$ ).



# Problemas de Clasificación

**Recuerdo:** Buscamos predecir *etiquetas*  $y \in \mathcal{C}$ , con  $\mathcal{C}$  finito (digamos,  $\mathcal{C} = [K]$ ).  
¿Cómo hacemos esta *finitud* compatible con nuestros modelos *diferenciables*?

# Problemas de Clasificación

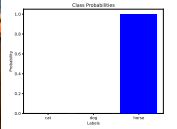
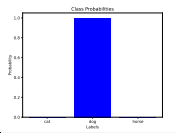
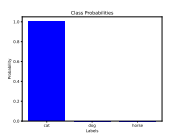
**Recuerdo:** Buscamos predecir *etiquetas*  $y \in \mathcal{C}$ , con  $\mathcal{C}$  finito (digamos,  $\mathcal{C} = [K]$ ).  
¿Cómo hacemos esta *finitud* compatible con nuestros modelos *diferenciables*?

Usamos **one-hot encodings** (OHE) para representar las labels: i.e. vemos  $y \in \mathcal{C}$  como un vector  $\vec{y} = (\mathbb{1}_{z=j})_{j=1}^K \in \Delta_K := \{\vec{u} \in [0, 1]^K : \sum_{j=1}^K u_j = 1\}$  (probabilidad).

# Problemas de Clasificación

**Recuerdo:** Buscamos predecir *etiquetas*  $y \in \mathcal{C}$ , con  $\mathcal{C}$  finito (digamos,  $\mathcal{C} = [K]$ ).  
¿Cómo hacemos esta *finitud* compatible con nuestros modelos *diferenciables*?

Usamos **one-hot encodings** (OHE) para representar las labels: i.e. vemos  $y \in \mathcal{C}$  como un vector  $\vec{y} = (\mathbb{1}_{z=y})_{j=1}^K \in \Delta_K := \{\vec{u} \in [0, 1]^K : \sum_{j=1}^K u_j = 1\}$  (probabilidad).



# Problemas de Clasificación

En general,  $\hat{y} = \Phi_{\theta}(x) \in \mathbb{R}^K$  es **no acotado**: para hacerlo comparable a un OHE, hay que **normalizarlo** (de forma *suave*).

# Problemas de Clasificación

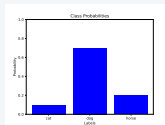
En general,  $\hat{y} = \Phi_{\theta}(x) \in \mathbb{R}^K$  es **no acotado**: para hacerlo comparable a un OHE, hay que **normalizarlo** (de forma *suave*).

Para ello utilizamos la función **softmax**:

$$\text{softmax}(\vec{y}) = \frac{\exp(\vec{y})}{\sum_{j=1}^K \exp(y_j)} \in \Delta_K$$



$\text{softmax}(\Phi_{\theta}(x))$

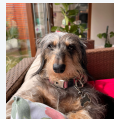


# Problemas de Clasificación

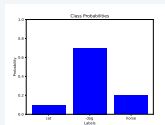
En general,  $\hat{y} = \Phi_{\theta}(x) \in \mathbb{R}^K$  es **no acotado**: para hacerlo comparable a un OHE, hay que **normalizarlo** (de forma *suave*).

Para ello utilizamos la función **softmax**:

$$\text{softmax}(\vec{y}) = \frac{\exp(\vec{y})}{\sum_{j=1}^K \exp(y_j)} \in \Delta_K$$



$\text{softmax}(\Phi_{\theta}(x))$



Esta es una versión *suave* del OHE/argmax, de hecho:

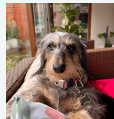
$$\text{argmax}(\vec{y}) = \lim_{\tau \rightarrow 0^+} \text{softmax}(\vec{y}/\tau)$$

# Problemas de Clasificación

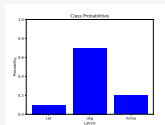
En general,  $\hat{y} = \Phi_{\theta}(x) \in \mathbb{R}^K$  es **no acotado**: para hacerlo comparable a un OHE, hay que **normalizarlo** (de forma *suave*).

Para ello utilizamos la función **softmax**:

$$\text{softmax}(\vec{y}) = \frac{\exp(y_j)}{\sum_{j=1}^K \exp(y_j)} \in \Delta_K$$



$\text{softmax}(\Phi_{\theta}(x))$



Esta es una versión *suave* del OHE/argmax, de hecho:

$$\text{argmax}(\vec{y}) = \lim_{\tau \rightarrow 0^+} \text{softmax}(\vec{y}/\tau)$$

Ahora, ¿Cómo evaluamos la similitud entre las probabilidades reales y predichas?

# Problemas de Clasificación

Definamos:



# Problemas de Clasificación

Definamos:

- **Entropía:**  $H(\vec{p}) := -\sum_{j=1}^K p_j \log p_j = \mathbb{E}_{\vec{p}}[-\log p.]$

# Problemas de Clasificación

Definamos:

- **Entropía:**  $H(\vec{p}) := -\sum_{j=1}^K p_j \log p_j = \mathbb{E}_{\vec{p}}[-\log p.]$
- **Entropía Cruzada:**  $\text{CE}(\vec{p}, \vec{q}) := -\sum_{j=1}^K p_j \log q_j =: \mathbb{E}_{\vec{p}}[-\log q.] \geq H(\vec{p})$

# Problemas de Clasificación

Definamos:

- **Entropía:**  $H(\vec{p}) := -\sum_{j=1}^K p_j \log p_j = \mathbb{E}_{\vec{p}}[-\log p.]$
- **Entropía Cruzada:**  $\mathbf{CE}(\vec{p}, \vec{q}) := -\sum_{j=1}^K p_j \log q_j =: \mathbb{E}_{\vec{p}}[-\log q.] \geq H(\vec{p})$
- **Kullback-Leibler Divergence:**  $D_{KL}(\vec{p}||\vec{q}) := \mathbf{CE}(\vec{p}, \vec{q}) - H(\vec{p}) \geq 0$

# Problemas de Clasificación

Definamos:

- **Entropía:**  $H(\vec{p}) := -\sum_{j=1}^K p_j \log p_j = \mathbb{E}_{\vec{p}}[-\log p.]$
- **Entropía Cruzada:**  $\mathbf{CE}(\vec{p}, \vec{q}) := -\sum_{j=1}^K p_j \log q_j =: \mathbb{E}_{\vec{p}}[-\log q.] \geq H(\vec{p})$
- **Kullback-Leibler Divergence:**  $D_{KL}(\vec{p}||\vec{q}) := \mathbf{CE}(\vec{p}, \vec{q}) - H(\vec{p}) \geq 0$

Pese a no ser medidas de **distancia**, las últimas 2 son ampliamente usadas en la práctica como pérdidas para el problema de clasificación (la famosa **Cross-Entropy Loss**, que es diferenciable y fácil de calcular).

# Problemas de Clasificación

Definamos:

- **Entropía:**  $H(\vec{p}) := -\sum_{j=1}^K p_j \log p_j = \mathbb{E}_{\vec{p}}[-\log p.]$
- **Entropía Cruzada:**  $\mathbf{CE}(\vec{p}, \vec{q}) := -\sum_{j=1}^K p_j \log q_j =: \mathbb{E}_{\vec{p}}[-\log q.] \geq H(\vec{p})$
- **Kullback-Leibler Divergence:**  $D_{KL}(\vec{p}||\vec{q}) := \mathbf{CE}(\vec{p}, \vec{q}) - H(\vec{p}) \geq 0$

Pese a no ser medidas de **distancia**, las últimas 2 son ampliamente usadas en la práctica como pérdidas para el problema de clasificación (la famosa **Cross-Entropy Loss**, que es diferenciable y fácil de calcular). Es más:

Teorema (CELoss es EMV en clasificación)

*El minimizador  $\theta^* \in \operatorname{argmin} \left\{ \hat{C}(\theta) := -\frac{1}{m} \sum_{i=1}^m \log p_{\theta}(z_i) \right\}$ , es el EMV del problema.*

# Problemas con Estructura

En ciertos de los problemas que resolvemos, hay una cierta *estructura* subyacente (simetrías, invarianza, etc.) que podemos intentar aprovechar.

# Problemas con Estructura

En ciertos de los problemas que resolvemos, hay una cierta *estructura* subyacente (simetrías, invarianza, etc.) que podemos intentar aprovechar.



# Problemas con Estructura

En ciertos de los problemas que resolvemos, hay una cierta *estructura* subyacente (simetrías, invarianza, etc.) que podemos intentar aprovechar.



¿Cómo introducimos este *sesgo inductivo* en nuestros modelos de NN?



# Redes Neuronales Convolucionales

Como ejemplo, consideramos la **clasificación de imágenes** (e.g. MNIST).

# Redes Neuronales Convolucionales

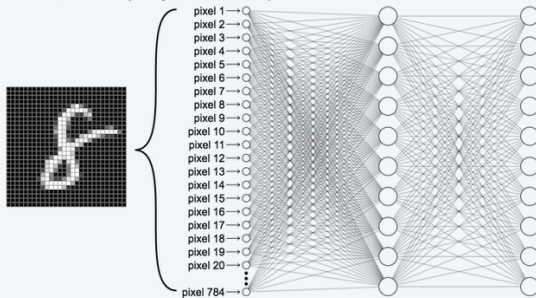
Como ejemplo, consideramos la **clasificación de imágenes** (e.g. MNIST).

- Si queremos atacarlo con FFNNs, debemos *aplanar* los datos y estudiar **cada pixel** independientemente, sin considerar su distribución espacial:

# Redes Neuronales Convolucionales

Como ejemplo, consideramos la **clasificación de imágenes** (e.g. MNIST).

- Si queremos atacarlo con FFNNs, debemos *aplanar* los datos y estudiar **cada pixel** independientemente, sin considerar su distribución espacial:
  - e.g. vemos una imagen de  $28 \times 28$  como un vector en  $\mathbb{R}^{784}$
  - $\uparrow$  N° de parámetros,  $\uparrow$  complejidad, 0 aprovechamiento de la estructura.



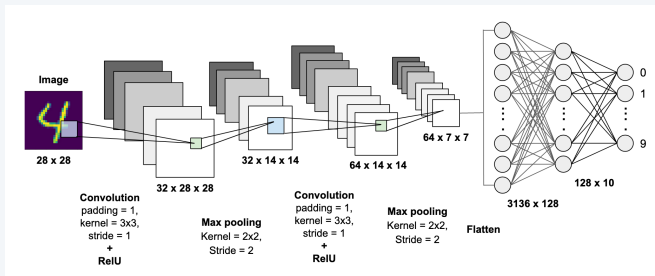
# Redes Neuronales Convolucionales

Como ejemplo, consideramos la **clasificación de imágenes** (e.g. MNIST).

# Redes Neuronales Convolucionales

Como ejemplo, consideramos la **clasificación de imágenes** (e.g. MNIST).

- Las Redes Neuronales Convolucionales (CNN) aparecen para aprovechar la estructura *traslacional* de las imágenes. Se **comparten pesos** y tienen una estructura **sparse** de conexiones (i.e.  $\downarrow$  N° parámetros, ).



## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$

## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature

1	1 <sub>x1</sub>	1 <sub>x0</sub>	0 <sub>x1</sub>	0
0	1 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	0
0	0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	

Convolved  
Feature

## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	4

Convolved  
Feature

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4	3	4
2		

Convolved  
Feature



## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$

1	1	1	0	0
0	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0
0	0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1
0	0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0
0	1	1	0	0

Image

4	3	4
2	4	

Convolved  
Feature

1	1	1	0	0
0	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0
0	0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x0</sub>
0	0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0 <sub>x1</sub>
0	1	1	0	0

Image

4	3	4
2	4	3

Convolved  
Feature

## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$

1	1	1	0	0
0	1	1	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0 <sub>x0</sub>	0 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0

Image

4	3	4
2	4	3
2		

Convolved  
Feature

1	1	1	0	0
0	1	1	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x1</sub>	1
0 <sub>x0</sub>	0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x0</sub>	0
0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0 <sub>x1</sub>	0

Image

4	3	4
2	4	3
2	3	

Convolved  
Feature

## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$

1	1	1	0	0
0	1	1	1	0
0	0	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>
0	0	1 <sub>x0</sub>	1 <sub>x1</sub>	0 <sub>x0</sub>
0	1	1 <sub>x1</sub>	0 <sub>x0</sub>	0 <sub>x1</sub>

Image

4	3	4
2	4	3
2	3	4

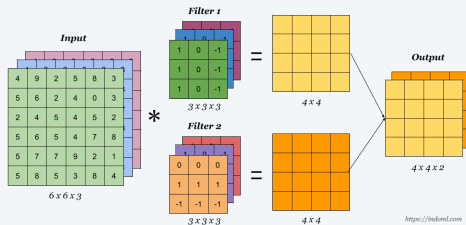
Convolved  
Feature

- El *stride* define, adicionalmente, cada cuánto se aplica el kernel.
- En el caso finito, el *padding* define un rango, en torno a la imagen, que se rellena con 0.

## Definición (Convolución 2D; idea generalizable)

Para una imagen de entrada  $x = (x_{(i,j) \in \Sigma})$  y *filtro*  $w = (w_{(i,j) \in \Sigma})$ , con  $\Sigma$  siendo  $\mathbb{Z}_N \times \mathbb{Z}_M$  (finito) o  $\mathbb{Z}^2$  (bi-infinito); definimos la operación de **convolución** ( $C_w(x) := x * w$ ):

$$(x * w)_{i,j} = \sum_{(k,\ell) \in \Sigma} x_{i+k,j+\ell} w_{k,\ell}, \quad (i,j) \in \Sigma$$



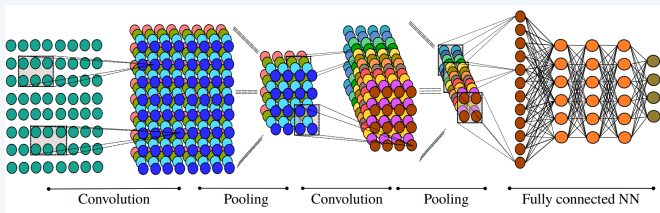
- En realidad, se aplican **varios filtros en paralelo** a una imagen.
- Como la convolución es *lineal*, se puede entender como una multiplicación de matrices *especiales* (*parameter-sharing*).

## Definición (Red Neuronal Convolutacional)

Una **CNN** consiste en la composición de *bloques de convolución*, que contienen: una capa de **convolución**, seguida de una ReLU y una operación de pooling (e.g. MaxPool, AvgPool, etc.). El resultado de los bloques se *aplana* y se pasa por una FFNN final.

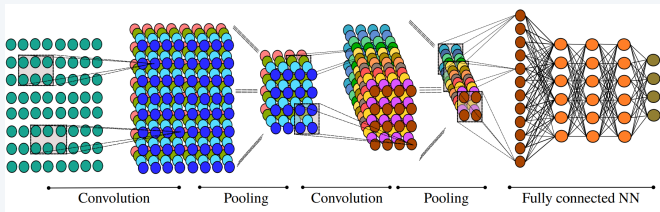
## Definición (Red Neuronal Convolucional)

Una **CNN** consiste en la composición de *bloques de convolución*, que contienen: una capa de **convolución**, seguida de una ReLU y una operación de pooling (e.g. MaxPool, AvgPool, etc.). El resultado de los bloques se *aplana* y se pasa por una FFNN final.





## Definición (Red Neuronal Convolucional)

Una **CNN** consiste en la composición de *bloques de convolución*, que contienen: una capa de **convolución**, seguida de una ReLU y una operación de pooling (e.g. MaxPool, AvgPool, etc.). El resultado de los bloques se *aplana* y se pasa por una FFNN final.



Los kernel se *entrenan* para extraer características *locales* relevantes para la tarea.

# References

-  Calin, Ovidiu. (2020).  
“Deep Learning Architectures: A Mathematical Approach,” 1st ed. 2020.  
*Springer International Publishing.*
-  Grohs, P., Kutyniok, G., eds. (2022)  
*Mathematical Aspects of Deep Learning,*  
Cambridge University Press.



**CMM**

Center for  
Mathematical  
Modeling

**¡Gracias por su atención!**

**Diego Olguín (basado en un trabajo conjunto con Javier Maass)**

Centro de Modelamiento Matemático  
Universidad de Chile

9 de septiembre